Delft University of Technology

On The Relation of Test Smells to Software Code Quality

Spadini, Davide; Palomba, Fabio; Zaidman, Andy; Bruntink, Magiel; Bacchelli, Alberto

DOI
10.1109/ICSME.2018.00010

Publication date
2018

Document Version
Peer reviewed version

Published in
34th International Conference on Software Maintenance and Evolution (ICSME'18)

Citation (APA)
Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., & Bacchelli, A. (2018). On The Relation of Test Smells to Software Code Quality. In 34th International Conference on Software Maintenance and Evolution (ICSME'18) (pp. 12-23). IEEE. DOI: 10.1109/ICSME.2018.00010

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

On The Relation of Test Smells to Software Code Quality

Davide Spadini,*‡ Fabio Palomba§ Andy Zaidman,* Magiel Bruntink,‡ Alberto Bacchelli§
‡Software Improvement Group, *Delft University of Technology, §University of Zurich
*{d.spadini, a.e.zaidman}@tudelft.nl, ‡m.bruntink@sig.eu, §{palomba, bacchelli}@ifi.uzh.ch

Abstract—**Test smells are sub-optimal design choices in the implementation of test code. As reported by recent studies, their presence might not only negatively affect the comprehension of test suites but can also lead to test cases being less effective in finding bugs in production code. Although significant steps toward understanding test smells, there is still a notable absence of studies assessing their association with software quality.**

In this paper, we investigate the relationship between the presence of test smells and the change- and defect-proneness of test code, as well as the defect-proneness of the tested production code. To this aim, we collect data on 221 releases of ten software systems and we analyze more than a million test cases to investigate the association of six test smells and their co-occurrence with software quality. Key results of our study include:(i) tests with smells are more change- and defect-prone, (ii) 'Indirect Testing', 'Eager Test', and 'Assertion Roulette' are the most significant smells for change-proneness and, (iii) production code is more defect-prone when tested by smelly tests.

I. Introduction

Automated testing (hereafter referred to as just *testing*) has become an essential process for improving the quality of software systems [12], [47]. In fact, testing can help to point out defects and to ensure that production code is robust under many usage conditions [12], [16]. Writing tests, however, is as challenging as writing production code and developers should maintain test code with the same care they use for production code [11].

Nevertheless, recent studies found that developers perceive and treat production code as more important than test code, thus generating quality problems in the tests [9], [10], [57], [82]. This finding is in line with the experience reported by van Deursen *et al.* [74], who described how the quality of test code was "not as high as the production code [because] test code was not refactored as mercilessly as our production code" [74]. In the same work, van Deursen *et al.* introduced the concept of *test smells*, inspired by Fowler *et al.* 's *code smells* [23]. These smells were recurrent problems that van Deursen *et al.* found when refactoring their troublesome tests [45].

Since its inception, the concept of test smells has gained significant traction both among practitioners [18], [42] and the software engineering research community [7], [26], [74], [76]. Bavota *et al.* presented the earliest and most significant results advancing our empirical knowledge on the effects of test smells [7]. The researchers conducted the first controlled laboratory experiment to establish the impact of test smells on program comprehension during maintenance activities and found evidence of a negative impact of test smells on both comprehensibility and maintainability of test code [7].

Although the study by Bavota *et al.* [7] made a first, necessary step toward the understanding of maintainability aspects of test smells, our empirical knowledge on whether and how test smells are associated with software quality aspects is still limited. Indeed, van Deursen *et al.* [74] based their definition of test smells on their anecdotal experience, without extensive evidence on whether and how such smells are negatively associated with the overall system quality.

To fill this gap, in this paper we quantitatively investigate the relationship between the presence of smells in test methods and the change- and defect-proneness of both these test methods and the production code they intend to test. Similar to several previous studies on software quality [24], [62], we employ the proxy metrics change-proneness (*i.e.*, number of times a method changes between two releases) and defect-proneness (*i.e.*, number of defects the method had between two releases). We conduct an extensive observational study [15], collecting data from 221 releases of ten open source software systems, analyze more than a million test cases, and investigate the association between six test smell types and the aforementioned proxy metrics.

Based on the experience and reasoning reported by van Deursen *et al.* [74], we expect to find tests affected by smells to be associated with more changes and defects, *i.e.*, higher maintenance efforts and lower software quality. Furthermore, since test smells indicate poor design choices [74] and previous studies showed that better test code quality leads to better productivity when writing production code [4], we expect to find production code tested by smelly tests to be associated with more defects.

Our results meet these expectations: Tests with smells are more change- and defect-prone than tests without smells and production code is more defect-prone when tested by smelly tests. Among the studied test smells, 'Indirect testing', 'Eager Test' and 'Assertion Roulette' are those associated with highest change-proneness; moreover, the first two are also related to a higher defect-proneness of the exercised production code. Overall, our results provide empirical evidence that detecting test smells is important to signal underlying software issues as well as studying the interplay between test design quality and effectiveness on detecting defects is of paramount importance for the research community.

II. Related Work

Over the last decade the research community spent a considerable effort in studying (*e.g.*, [1], [3], [32], [39], [51], [55], [59], [61], [66], [72], [78]–[80]) and detecting (*e.g.*, [33], [36], [41], [43], [46], [49], [52], [54], [70]) design flaws occurring in production code, also known as *code smells* [23]. At the same time, problems concerning the design of *test code* have only been partially explored and our literature survey showed us that our empirical knowledge is still limited.

In this section, we first discuss the literature related to test smells, then we discuss previous work that analyzed the change- and defect-proneness of code smells, as it can shed light on why test smells can also be problematic.

A. Test Smells

The importance of having well-designed test code was initially put forward by Beck [8]. Beck argued that test cases respecting good design principles are desirable since these test cases are easier to comprehend, maintain, and can be successfully exploited to diagnose problems in the production code. Inspired by these arguments, van Deursen *et al.* [74] coined the term test smells and defined the first catalog of 11 poor design choices to write tests, together with refactoring operations aimed at removing them. Such a catalog has been then extended more recently by practitioners, such as Meszaros [42] who defined 18 new test smells.

From these catalogs, Greiler *et al.* [25], [26] showed that test smells affecting test fixtures frequently occur in a company setting. Motivated by this prominence, Greiler *et al.* presented TESTHOUND, a tool able to identify fixture-related test smells such as 'General Fixture' or 'Vague Header Setup' [25]. Van Rompaey *et al.* [76] devised a heuristic code metric-based technique that can identify two test smell types, *i.e.*, 'General Fixture' and 'Eager Test'. However, the empirical study conducted to assess the performance of the technique showed that it often misses instances of the two smells.

Turning the attention to the empirical studies that had test smells as their object, Bavota *et al.* [7] studied (i) the diffusion of test smells in 18 software projects, and (ii) their effects on software maintenance. They found that 82% of JUnit classes are affected by at least one test smell and that the presence of test smells has a strong negative impact on the comprehensibility of the affected classes. The high diffuseness of test smells was also confirmed in the context of the test cases automatically generated by testing tools [53].

Tufano *et al.* [71] conducted an empirical study aimed at measuring the perceived importance of test smells and their lifespan during the software life cycle. Key results of the investigation indicated that developers usually introduce test smells in the first commit involving the affected test classes, and in almost 80% of the cases the smells are never removed, primarily because of poor awareness of developers. This study strengthened the case for having tools able to automatically detect test smells to raise developers' knowledge about these issues.

Finally, Palomba and Zaidman [56] investigated the extent to which test smells can be exploited to locate flaky tests, *i.e.*, test cases having a non-deterministic behavior [40]. The main findings of the work showed that (i) almost 54% of flaky tests contain a test smell that can cause the flakiness and (ii) the refactoring of test smells removed both the design flaws and test code flakiness [56].

The work we present in this paper is complementary to the ones discussed so far: We aim at making a further step ahead by investigating the change- and defect-proneness of test smells, as well as the defect-proneness of production code tested by smelly tests.

B. Change- and Defect-proneness of Code Smells

The software engineering research community has conducted extensive work in the context of code smells in production code. More specifically, Khomh *et al.* [31] showed that the presence of code smells increases the code's change-proneness. Later on, they also showed that code components affected by code smells are more fault-prone than non-smelly components [32]. Their results were confirmed by Palomba *et al.* [50], who found that code smells make classes more change- and defect-prone; in addition, they also found that the class' change-proneness can benefit from code smell removal, while the presence of code smells in many cases is not necessarily the direct cause of the class defect-proneness, but rather a co-occurring phenomenon [50].

Gatrell and Counsell [24] conducted an empirical study aimed at quantifying the effect of refactoring on class change- and defect-proneness. In particular, they monitored a commercial project for eight months and identified the refactoring operations applied by developers during the first four months. Then, they examined the same classes for the second four months to investigate whether the refactoring results in a decrease of change- and defect-proneness. They compared against classes of the system that were not refactored during the same period. Results revealed that classes subject to refactoring have a lower change- and defect-proneness.

Li and Shatnawi [38] empirically evaluated the correlation between the presence of code smells and the probability that the class contains errors. They studied the post-release evolution process showing that many code smells are positively correlated with class errors. Olbrich *et al.* [48] studied the maintainability of two specific code smell types, *i.e.*, 'God Class' and 'Brain Class', reporting that classes affected by such smells change less frequently and have a fewer number of defects than non-smelly classes. D'Ambros *et al.* [20] studied how 'Feature Envy' and 'Shotgun Surgery' instances are related to software defects, reporting no consistent correlation between them. Finally, Saboury *et al.* [63] empirically investigated the impact of code smells on the defect-proneness of JAVASCRIPT modules, confirming the adverse effect of smells on source code maintainability.

III. Research Methodology

The *goal* of our study is to increase our empirical knowledge on whether and how test methods affected by smells are associated with higher change- and defect- proneness of the test code itself, as well as to assess whether and to what extent test methods affected by test smells are associated with the defect-proneness of the production code they test. The *perspective* is that of both researchers and practitioners who are interested in understanding the possible adverse effects of test smells on test and production code. We structured our study around the two overarching research questions that we describe in the following.

The first research question investigates the relationship between the presence of test smells in test code and its change/defect proneness:

RQ1. *Are test smells associated with change/defect proneness of test code?*

We, thus, structure **RQ1** in three sub-research questions. First, we aim at providing a broad overview of the relationship of test smells and their co-occurrence with change- and defect-proneness of test code:

RQ1.1: *To what extent are test smells associated with the change- and defect- proneness of test code?*

RQ1.2: *Is the co-occurrence of test smells associated with the change- and defect-proneness of test code?*

Then, we aim at verifying whether some particular test smells have a stronger association with change- and defect-proneness of test code:

RQ1.3: *Are certain test smell types more associated with the change- and defect-proneness of test code?*

Considering that defect-proneness as been widely used in previous literature as a proxy metric for software quality (*e.g.*, [20], [24], [32], [50]), in the second research question, we aim at making a complementary analysis into the association of test smells with the defect-proneness of the exercised production code. In fact, if the production code exercised by tests with test smells is more defect-prone this would be an even stronger signal on the relevance of test smells. This goal leads to our second research question:

RQ2. *Is the production code tested by tests affected by test smells more defect-prone?*

The expectation is that test code affected by test smells might be less effective in detecting defects [4], thus being associated with more defect-prone production code. We structured **RQ2** in three sub-research questions:

RQ2.1: *Are test smells associated with the defect-proneness of the tested production code?*

TABLE I
Subject systems' details

System	#Releases	#Classes (Min-Max)	#Methods (Min-Max)	#KLOC (Min-Max)
Apache Ant	10	9-282	74-2,541	1-25
Apache Cassandra	25	61-437	237-4,804	2-59
Apache Hadoop	35	470-1,895	3,400-19,445	71-344
Apache Wicket	44	102-585	587-3,351	8-46
Eclipse JDT	17	11-56	68-4,068	1-49
ElasticSearch	36	25-698	324-6,755	5-118
Hibernate	8	823-1,508	5,461-9,027	92-144
Sonarqube	36	492-2,072	2,256-18,028	18-134
Spring Framework	7	980-1,662	10,576-18,049	136-212
VRaptor4	3	122-125	1,046-1,102	8-9
Total	221	9-2,072	68-19,445	1-344

RQ2.2: *Is the co-occurrence of test smell associated with the defect-proneness of the tested production code?*

RQ2.3: *Are certain test smell types more associated with the defect-proneness of production code?*

Similarly to **RQ1**, we aim at providing an overview of the role of test smells in the defect-proneness of production code, by investigating single test smells and their co-occurrence.

A. Subjects of the Study

In our study, we have to select two types of subjects: software systems and test smells.

Software systems. We consider ten OSS projects and their 221 major releases as subject systems for our study. Specifically, Table I reports the characteristics of the analyzed systems concerning (i) the number of the considered releases and (ii) size, in terms of the number of classes, methods, and KLOCs. Two main factors drive the selection: firstly, since we have to run static analysis tools to detect test smells and compute maintainability metrics, we focus on projects whose source code is publicly available (*i.e.*, OSS); secondly, we analyze systems having different sizes and scopes. After filtering on these criteria, we randomly select ten OSS projects from the list available on GitHub[1] having different size, scope, and with a number of JUnit test cases higher than 1,000 in all the releases.

For each system, we only consider their major releases. In fact, (i) detecting test smells at commit-level is prohibitively expensive in terms of computational time and (ii) minor releases are too close to each other (in some cases there is more than one minor release per week), so very few changes are made in the source and test code. We mine these major releases directly from the systems' GitHub repositories.

Test smells. As subject test smells for our study, we consider those described in Table II. While other test smell types have been defined in literature [42], [74], we select the smells in Table II because: (1) Identifying test smells in 221 project releases through manual detection is prohibitively expensive, thus a reliable and accurate automatic detection mechanism must be available; (2) the selected test smells have the greatest diffusion in industrial and OSS projects [7]; and (3) the

[1]https://github.com

TABLE II
SUBJECT TEST SMELLS

Test smell	Description	Problem
'Mystery Guest'	A test that uses external resources (e.g., file containing test data)	Lack of information makes it hard to understand. Moreover, using external resources introduces hidden dependencies: if someone deletes such a resource, tests start failing.
'Resource Optimism'	A test that makes optimistic assumptions about the state/existence of external resources	It can cause non-deterministic behavior in test outcomes. The situation where tests run fine at one time and fail miserably the other time.
'Eager Test'	A test method exercising more methods of the tested object	It is hard to read and understand, and therefore more difficult to use as documentation. Moreover, it makes tests more dependent on each other and harder to maintain.
'Assertion Roulette'	A test that contains several assertions with no explanation	If one of the assertions fails, you do not know which one it is.
'Indirect Testing'	A test that interacts with the object under test indirectly via another object	This smell indicates that there might be problems with data hiding in the production code.
'Sensitive Equality'	A test using the 'toString' method directly in assert statements	It may depend on many irrelevant details such as commas, quotes, spaces, etc. Whenever the toString method for an object is changed, tests start failing.

selected ones compose a diverse catalog of test smells, which are related to different characteristics of test code.

B. Data Extraction

To answer **RQ1**, we extract data about (i) the test smells affecting the test methods in each system release and (ii) the change/defect proneness of these test cases. To answer **RQ2**, we extract data about the defect proneness of the production code exercised by the test code. The obtained data and the *R* script used to analyze the results are both available in our online appendix [14].

Detecting test smells. We adopt the test smell detector by Bavota *et al.* [7] (widely adopted in previous research [7], [53], [56], [71]), which is able to reliably identify the six smells considered in our study with a precision close to 88% and a recall of 100%, by relying on code metrics-based rules.

Defining the change-proneness of test code. To compute change- and defect-proneness of test code, we mine the change history information of the subject systems using REPODRILLER [2], a Java framework that allows the extraction of information such as commits, modifications, diffs, and source code. Explicitely, for each test method T_i of a specific release r_j we compute its change-proneness as follows:

$$change\_proneness(T_i, r_j) = \#commits(T_i)_{r_{j-1} \rightarrow r_j}$$

where $\#commits(T_i)_{r_{j-1} \rightarrow r_j}$ represents the number of changes performed by developers on the test method T_i between the releases r_{j-1} and r_j. Given the granularity of our analyses (*i.e.*, release-level), we only compute the change-proneness of test methods that were actually present in a release r_j; if a new method was added and removed between r_{j-1} and r_j, it does not appear in our result set. To identify which test method changed within a commit, we implement the following algorithm:

1) We first identify all test classes modified in the commit. In line with past literature [71], [81], we consider a class to be a test when its name ends with 'Test' or 'Tests'.
2) For each test class, we obtain the source code of the class in both the present commit and the previous one.
3) We parse the source code of the test class to identify the test methods contained in the current and in the previous commit. Then, we compare the source code of each test method from the current commit against all the test methods of the prior version:
   a) if we find the same method, it means that it is not changed (*i.e.*, both signature and content of the method in r_j are the same as r_{j-1});
   b) if we find a different method, it means that it is changed (*i.e.*, the signature of the method is the same, but the source code in r_j is not equal to r_{j-1});
   c) if we do not find the method (*i.e.*, the signature of the method does not exist in the previous version of the file), it means that it has been added or renamed. To capture the latter, we adopt a technique similar to the one proposed by Biegel *et al.* [13], based on the use of textual analysis to detect rename refactoring operations. Specifically, if the cosine similarity [5] between the current method and that of the methods in the previous version is higher than 95%, then we consider a method as renamed (hence, it inherited all the information of the old test case).

Defining the defect-proneness of test code. To compute the defect-proneness of each test case, we follow a similar procedure to the one for change-proneness, with the exception that to calculate the *buggy* commits we relied on SZZ [67]. In particular, we first determine whether a commit fixed a defect employing the technique proposed by Fischer *et al.* [22], which is based on the analysis of commit messages. If a commit message matches an issue ID present in the issue tracker or it contains keywords such as *'bug'*, *'fix'*, or *'defect'*, we consider it as a bug fixing activity. This approach has been extensively used in the past to determine bug fixing changes [29], [34] and it has an accuracy close to 80% [22], [55], thus we deem it as being accurate enough for our study. Once we have detected all the bug fixing commits involving a test method, we employ SZZ to obtain the commits where the bug was introduced.

To estimate the moment when a bug was likely introduced, the SZZ algorithm relies on the annotation/blame feature of versioning systems [67]. In short, given a bug-fix activity identified by the bug ID k, the approach works as follows:

- For each file f_i, $i = 1 \dots m_k$ involved in the bug-fix k (m_k is the number of files changed in the bug-fix k) and fixed in its revision *rel-fix*$_{i,k}$, we extracted the file revision just *before* the bug fixing (*rel-fix*$_{i,k} - 1$).
- Starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the bug k, we identified the production method M_j to which the changed line changed belongs. Furthermore, the `blame` feature of `Git` is used to identify the revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified using an island grammar parser [44]. This produces, for each production method M_j, a set of $n_{i,k}$ bug-inducing revisions *rel-bug*$_{i,j,k}$, $j = 1 \dots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for inducing a bug.

With the list of bug inducing commits involving every test method, we compute its defect-proneness in a release r_j as the number of bug inducing activities involving the method in the period between the releases r_{j-1} and r_j .

Defining the defect-proneness of production code. For each test method in the considered projects, we first need to retrieve what is the production method it exercises. For this, we exploit a traceability technique based on naming convention, *i.e.*, it identifies the methods under test by removing the string *'Test'* from the method name of the JUnit test method. This technique has been previously evaluated by Sneed [68] and by Van Rompaey and Demeyer [75], demonstrating the highest performance (both in terms of accuracy and scalability) with respect to other traceability approaches (*e.g.*, slicing-based approaches [60]).

Once we detect the links between test and production methods, we can compute the defect-proneness of such production methods. Since we calculate test smells at the release level (*i.e.*, we only have information regarding which test is smelly at the specific commit of the release), we have to detect how many defects production methods have within that particular release. To this aim, we rely again on the SZZ algorithm. To detect defects of production code in a specific release, we only consider bug fixing activities related to bugs introduced *before* the release date. More formally, we compute the fault-proneness of a production method M_i in a release r_j as the number of changes to M_i aimed at fixing a bug in the period between r_j and r_{j+1}, where the bug was introduced before the release date, in the period between r_{j-1} and r_j. The obtained list of bugs are the ones that were present in the system when it was released, hence not captured using tests.

By employing SZZ, we can approximate the time periods in which each production method was affected by one or more bugs. We exclude from our analysis all the bugs occurring in a production method M_i after the system was released, because in this case the test smell could have been solved before the introduction of the bug. We also exclude bug-introducing changes that were recorded after the bug was reported, since they represent false positives [19].

C. Data Analysis

To answer **RQ1**, we analyze the previously extracted information regarding test smells and change- and defect- proneness of test code. In particular, in the context of **RQ1.1**, we test whether JUnit test methods that contain a test smell are more likely to be change- or defect-prone. To this aim, we compute the Relative Risk (RR) [37], an index reporting the likelihood that a specific cause (in our case, the presence/absence of a test smell) leads to an increase in the amount a test case is subject to a particular property (in our case, number of changes or defects) [30], [58]. The RR is defined as the ratio of the probability of an event occurring in an exposed group (*e.g.*, the probability of smelly tests being defective), to the probability of the event occurring in a non-exposed group (*e.g.*, the probability of non-smelly tests being defective) and it is computed using the following equation:

$$RR = \frac{p_{\text{event when exposed}}}{p_{\text{event when not exposed}}}$$

A relative risk of 1 means that the event is equally likely in both samples. A RR greater than 1 indicates that the event is more likely in the first sample (*e.g.*, when the test is smelly), while a RR of less than 1 points out it is more likely in the second sample (*e.g.*, when the test is not smelly). We prefer using this technique rather than alternative statistical tests adopted in previous work (*e.g.*, analysis of box plots [50] or Odds Ratios [6], [32]) because of the findings reported in the statistic field that showed how this method (i) should be preferred when performing exploratory studies such as the one conducted herein [21], [83] and (ii) is equivalent to Odds Ratios analysis [64].

Change- and defect-proneness of JUnit test methods might also be due to other factors rather than the presence of a test smell. Indeed, Kitchenham *et al.* [35] found that both size and number of previous changes might influence the observations on the defect-proneness of source code; additionally, Zhou *et al.* [84], reported the role of size as possible confounding effect when studying the change-proneness of code elements. Based on the evidence above, we control our findings for change-proneness by computing the RR achieved considering the size of the test method in terms of lines of code (LOC). Moreover, we control the phenomenon of defect-proneness by considering LOC of test methods and number of times the method changed from the last release (*i.e.*, prior changes). More specifically, the aim is to understand whether the likelihood of a test case being smelly *and* more change- or defect-prone varies when controlling for size and number of changes. In other words, if smelly tests are consistently more prone to changes and defects than non-smelly tests, independently from their size or number of times they changed in the past, we have higher confidence that the phenomena observed are associated with test smells.

To answer **RQ1.2** and analyze the role of test smell co-occurrences, we split the previously extracted dataset into seven groups, each one containing test methods affected by exactly i smells, where $0 \leq i \leq 6$. Then, we compare change- and defect-proneness of each group using (i) the Wilcoxon rank sum test [77] (with confidence level 95%) and (ii) Cohen's d [65] to estimate the magnitude of the observed difference. We choose the Wilcoxon test since it is a non-parametric test (it does not have any assumption on the underlying data distribution), while we interpret the results of Cohen's d relying on widely adopted guidelines [65]: The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and large for $d \geq 0.8$.

To answer **RQ1.3**, we adopt the same procedure as for **RQ1.2**, but we consider each smell type separately, *i.e.*, we compare change- and defect-proneness of different smell types by means of Wilcoxon rank sum test [77] and Cohen's d [65], controlling for size and number of previous changes (only in case of defect-proneness). It is important to note that, as done in earlier work [32], [50], in this analysis we consider test cases affected *only* by a single test smell, *e.g.*, only *Eager test*, with the aim of understanding the effect of single test smells on change- and fault-proneness of test code.

For **RQ2** we adopt a process similar to that of **RQ1**. In particular, for **RQ2.1** we compute the RR: in this case, we aim to investigate the likelihood that the presence/absence of a test smell is associated with the defect-proneness of the production code being tested. Similarly to **RQ1**, we control for size and number of changes. Analogously, in **RQ2.2** we use (i) the Wilcoxon rank sum test [77] and (ii) Cohen's d [28] to assess the association of test smell co-occurrences to the defect-proneness of production code. Finally, to answer **RQ2.3**, we compare the distribution of the number of defects related to the production code tested by different test smell types (considering *single* test smell types).

D. Threats to Validity

Our research method poses some threats to the validity of the results we obtain.

Construct validity. Threats to construct validity concern our research instruments. To obtain information regarding test smells we use the test smell detector devised by Bavota *et al.* [7]. Even though this tool has been assessed in previous studies [7], [56] as being extremely reliable, some false positives can still be present in our dataset.

Another threat is related to how we detected which production method is exercised by a test method: specifically, we exploited a traceability technique based on naming convention that has been heavily adopted in the past [6], [53], [71], [81]. This technique has also been evaluated by Sneed [68] and by Van Rompaey and Demeyer [75], and the results reported an average precision of 100% and a recall of 70%.

Internal validity. Threats to internal validity concern factors that could affect the variables and the relations being investigated. When we look into the relation between test smells and test defects, many factors can influence the results. For example, a test could contain more defects than others because more complex, bigger, or more coupled, while the studied variable (test smells) could be insignificant. To mitigate this, we control for some of these metrics, namely size of the method (LOC) and number of changes, which have been reported to correlate with code complexity [17]. As shown in the results section, the results generally do not change when controlling for other metrics. Furthermore, at the beginning of this study we also built a Logistic Regression Model to detect whether our explanatory variable was (not) statistically significant in the model. Similarly to Thongtanunam *et al.* [69], we built a logistic regression model to determine the likelihood of a test being defective (or change prone) using LOC, prior changes, production changes as control variables and being smelly (our new variable) as a binary explanatory variable. We used R scripts provided by Thongtanunam *et al.* [69] to build the model, and we discovered that test code smelliness was indeed statistically significant for the model. However, we preferred to proceed with RR instead of the model, for better readability of the results.

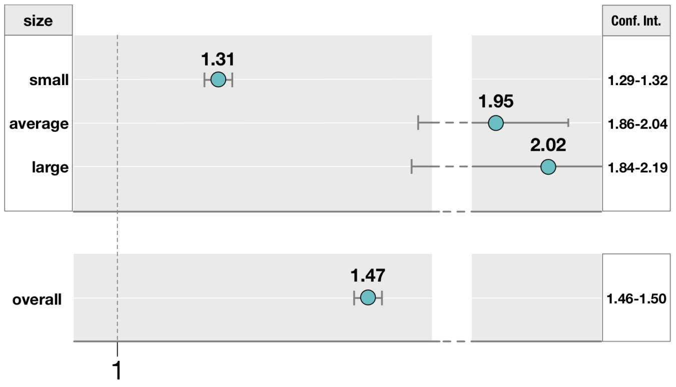

Fig. 1. Relative risk of being change prone in smelly tests *vs* non-smelly tests, controlling by size. The p-value for all RRs is < 0.0001.

External validity. Threats to external validity concern the generalization of results. We conducted our study taking into account 221 releases of 10 Java systems having different scope and characteristics to strengthen the generalizability of our findings. However, a study investigating different projects and programming languages may lead to differing conclusions.

IV. RQ1 Results: Test Smells and Test Code

This section describes the results to **RQ1**.

RQ1.1: To what extent are test smells associated with the change- and defect- proneness of test code?

Figure 1 depicts the Relative Risk of test smells to be associated with higher change-proneness of test cases (label "Overall") as well as how the risk is connected with the control factor analyzed, *i.e.*, size. In particular, we show how RR varies when the test method has (i) small size ($LOC < 30$), (ii) average size ($30 < LOC < 60$), and (iii) large size ($LOC > 60$). The thresholds used to identify small, medium, and large test methods were identified by applying the *Maintainability Model* proposed by Heitlager *et al.* [27], which cuts the distribution of all the method LOCs at the 70th, 80th and 90th percentiles. We also represent the p-value and the confidence interval for each category.

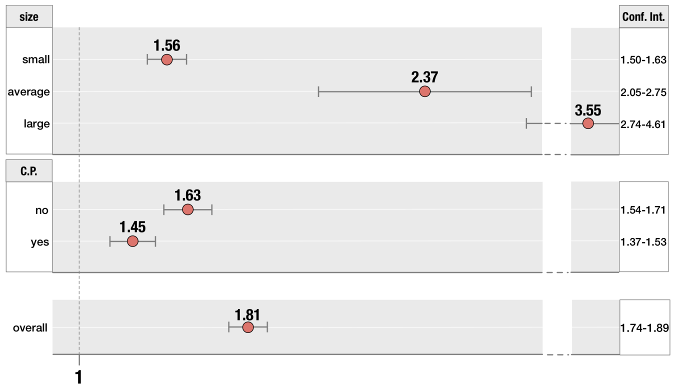

Fig. 2. Relative risk of being defect prone in smelly tests *vs* non-smelly tests, controlling by size and change proneness. For all RRs, p-value < 0.0001.

We make two main observations from the results in Figure 1. On the one hand, test methods affected by at least one smell are more change-prone than non-smelly methods, with an RR of 1.47; from a practical perspective, this means that a smelly test has the risk of being 47% more change-prone than a non-smelly test. On the other hand, we can notice that smelly tests with higher size are more change prone: this is intuitive since larger methods are more difficult to maintain (hence more change prone) and they are more likely to contain smells. An important result to notice is that large smelly tests ($LOC > 60$) are more than twice more likely of being change prone than not smelly large tests. This finding is a good incentive for practitioners and developers to write small and concise tests, as recommended by Beck [8].

Concerning defect-proneness, Figure 2 shows how the RR varies when considering (i) the presence of test smells ("Overall"), (ii) the size of test cases—split in the same way as done for change-proneness, and (iii) the number of previous changes applied to test cases (we discriminated between methods that change frequently vs. methods that infrequently change, by adopting the heuristic proposed by Romano and Pinzger [62], *i.e.*, we considered frequently evolving methods to have a number of changes higher than the median of the distribution of all the changes that occurred in test cases — 2, in our case).

From Figure 2, we observe that the presence of test smells is associated with the defect-proneness of test cases. Indeed, methods affected by at least one design flaw have the risk of being 81% more defect-prone than non-smelly ones. Additionally, the result does not change when controlling for size and number of changes. Indeed, the difference is even more prominent for large tests: the smelly ones are 3.5 times more defect prone than the not smelly. Instead, change proneness seems not relevant when discriminating the defect-proneness of test cases. In both cases, the RR of smelly tests of being more defect prone is 50% higher.

Overall, the results of this first analysis provide empirical evidence that test smells—defined with the aim of describing a set of bad patterns influencing test code maintainability [74]—are indeed associated with higher change- and defect-proneness of the affected test cases.

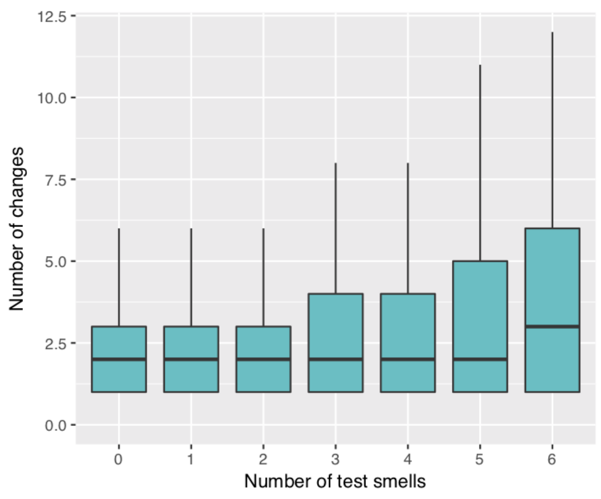

Fig. 3. Number of smells in a test method and corresponding number of changes to the method.

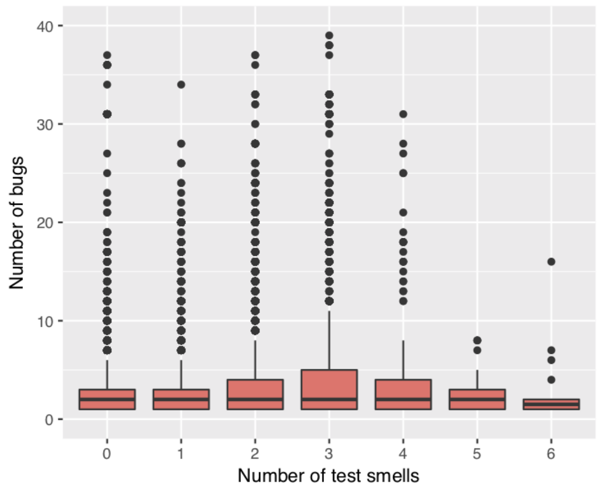

Fig. 4. Number of smells and number of defects

> ***Finding 1****. Tests affected by test smells are associated with higher change- and defect-proneness than tests not affected by smells, also when controlling for both the test size and the number of previous changes.*

RQ1.2 Is the co-occurrence of test smells associated with the change- and defect-proneness of test code?

While in the previous research question we did not discriminate on the number of test smells a test method contained, the goal of this analysis is to assess whether test smell co-occurrences is associated with the change- and defect-proneness of test cases. Figures 3 and 4 report box plots showing change- and defect-proneness of test cases affected by a different number of test smells, respectively.

For change-proneness, the median of the different groups very low (around one) for all test cases: to some extent, this is in line with the findings by Zaidman *et al.* [82], who found that developers generally do not change test cases as soon as they implement new modifications to the corresponding production

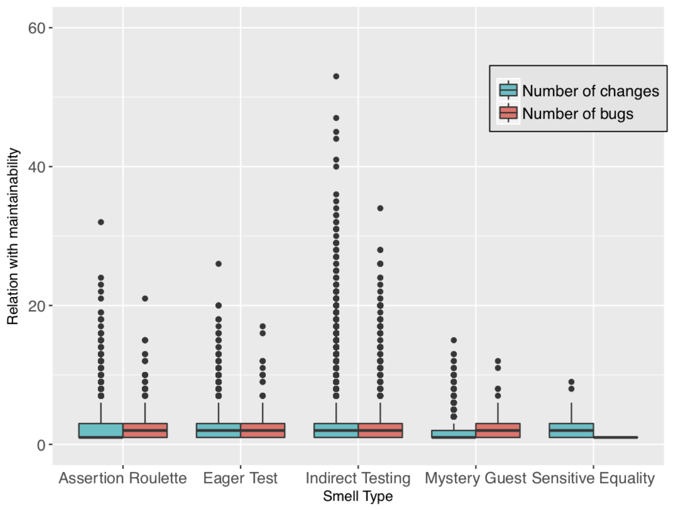

Fig. 5. Change- and fault- proneness of test methods affected by different types of smells.

code. At the same time, Figure 3 shows that the higher the number of test smells, the more dispersed the distribution of changes is, thus indicating that test cases affected by more design problems tend to be changed more often by developers. This observation is supported by the results of the statistical tests, where we found that the difference between all groups was statistically significant ($p - value < 2^{e-16}$), with a negligible effect size between the first 5 groups ($d \leq 0.2$) and a medium one between the first 5 and the last 2 groups ($0.5 \leq d \leq 0.8$).

When considering defect-proneness in Figure 4, we notice that test methods having up to four test smells do not show significant differences with respect to methods affected by five or six design flaws. Indeed, the median of the distribution is almost identical in all the groups, and even though the difference is considered statistically significant by the Wilcoxon rank sum test, it has a small effect size ($d < 0.2$). Thus, these findings suggest that the co-occurrence of more test smells is not directly associated with higher defect-proneness; we hypothesize that they are instead a co-existing phenomenon, similarly to what Palomba *et al.* reported for code smells in production code [50].

In the context of this research question, we controlled for the size of the test method and the number of its changes, finding that these factors are not associated with the investigated outcome. We include a report of this additional analysis in our on-line appendix [14].

***Finding 2**. Test methods affected by more smells are associated with a slightly higher change-proneness than methods with less smells. Conversely, the co-presence of more test smells in a test method is not associated with higher defect-proneness.*

RQ1.3 Are certain test smell types more associated with the change- and defect-proneness of test code?

The final step of the first research question investigates the association to change- and defect-proneness of different test smell types. Figure 5 shows two box plots for each type, depicting its change- and defect-proneness. When analyzing the change-proneness, we observe that almost all the test smells have a similar trend and indeed the magnitude of their differences is negligible, as reported by Cohen d. The only exception regards the *Indirect testing* smell: while the median change-proneness is similar to other smells, its box plot shows several outliers going up to 55 changes. In this case, the magnitude of the differences with all the other smell types is medium. This result is due to the characteristics of the smell. By definition, an *Indirect testing* smell is present when a method performs tests on other objects (*e.g.*, because of external references in the production code tested) [74]: as a consequence, it naturally triggers more changes since developers may need to modify the test code more often due to changes occurring in the exercised external production classes.

In the case of defect-proneness the discussion is similar. Indeed, the number of defects affecting the different test smell types is similar: even though the differences between them are statistically significant ($p\text{-}value < 2^{e-16}$), they are mostly negligible. However, we can see some exceptions, also in this case. The box plots show that the distribution of ‘Indirect Testing’, ‘Eager Test’ and ‘Assertion Roulette’ smells slightly differ from the others, and indeed these are the smells having the highest number of outliers. This result is due to the fact that these test smells tend to test more than required [74] (*i.e.*, a test method suffering from ‘Indirect Testing’ exercises other objects indirectly, an ‘Eager Test’ test method checks several methods of the object to be tested, while an ‘Assertion Roulette’ contains several assertions checking different behavior of the exercised production code). Their nature makes them intrinsically more complex to understand [7], likely leading developers to be more prone to introduce faults.

***Finding 3**. Test methods affected by ‘Indirect Testing’, ‘Eager Test’, and ‘Assertion Roulette’ are more change and defect prone than those affected by other smells.*

V. RQ2 Results: Test Smells and Production Code

This section describes the results to our second research question.

RQ2.1 Are test smells associated with the defect-proneness of the tested production code?

Figure 6 reports the RR that a smelly test case is exercising a more defect-prone production method (label ‘Overall’), along with the RR obtained when considering size as a control factor.

In the first place, Figure 6 shows that smelly tests have a higher likelihood to test defective code than non-smelly tests (*i.e.*, the RR = 1.71 states that production code executed by smelly tests has 71% higher chances of being defective than

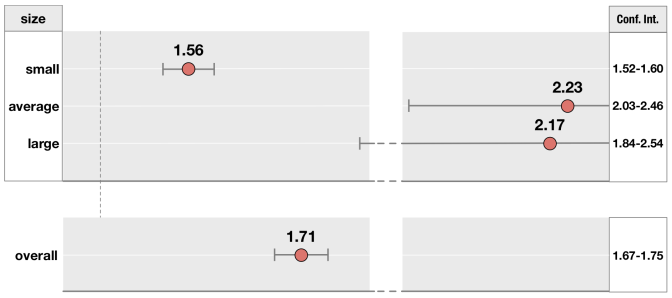

Fig. 6. Relative risk of the production code being more defect prone when tested by smelly tests *vs.* non-smelly tests. For all RRs, p-value < 0.0001.

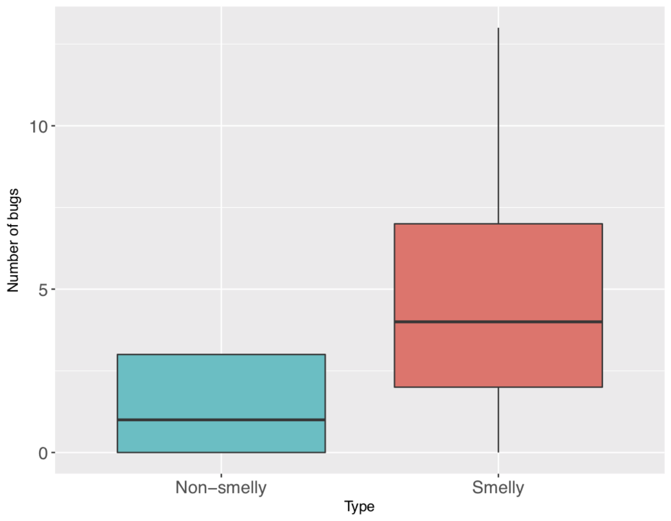

Fig. 7. Relative risk of being defect prone if tested by smelly tests *vs* non-smelly tests.

production code executed by non-smelly tests). Zooming in on this result, Figure 7 depicts the box plots reporting the distribution of the number of production code bugs, when exercised by smelly test methods vs. non-smelly ones. The difference between the two distributions is statistically significant ($p\text{-}value < 2.2^{e-16}$) with a large effect size ($d = 1.40$).

The results still hold when controlling for size: Size does not impact the RR concerning the defect-proneness of production code exercised by smelly tests vs. non-smelly ones, actually, as shown in the previous RQ, it makes it worst. For instance, methods having a large number of lines of code have an $RR = 2.17$. Two main factors can explain this result: On the one hand, we suppose that a large size of the test implies a large volume of the production code, and our research community widely recognized size as a valid proxy measure for software quality [35]; on the other hand, our results corroborate previous findings reported by Palomba *et al.* [50], who showed that large methods (*e.g.*, the ones affected by a *Long Method* code smell [23]) are strongly associated with the defect-proneness of production code.

Thus, from our analysis we have empirical evidence that the presence of test smells contributes to the explanation of the defect-proneness of production code. Given our experimental setting, we cannot speculate on the motivations behind the results achieved so far: indeed, our **RQ2.1** meant to be a coarse-grained investigation aimed at understanding whether the presence of design flaws in test code might somehow be associated with the defectiveness of production code. Thus, in this research question we did not focus on the reasons behind the relationship, *i.e.*, if it holds because the production code is of poor quality (thus difficult to test) or because the tests are of poor quality (thus they do not capture enough defects). Our **RQ2.3** makes a first step in providing additional insights on such a relationship.

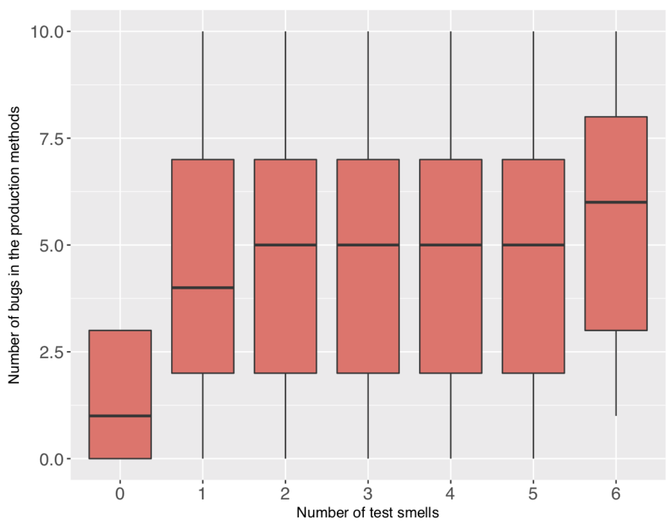

Fig. 8. Number of smells and number of production defects.

Finding 4*. Production code that is exercised by test code affected by test smells is more defect-prone, also when controlling for size.*

RQ2.2 Is the co-occurrence of test smell associated with the defect-proneness of the tested production code?

Figure 8 presents the results concerning the association of test smell co-occurrences to the defectiveness of the exercised production code. In this case, the defect-proneness of production code remains almost constant among the different groups, meaning that having more design issues in test code is not associated with a higher number of defects in production.

This result led to two main observations: as observed in **RQ2.1**, test smells are related to the defect-proneness of the exercised production code, but do not fully explain this phenomenon. Secondly, while the specific number of test smells is not associated with the defectiveness of production code, the overall presence of test smells is. It is reasonable to think that some *specific* test smells could contribute more to the found association to defect-proneness; this reasoning represented the input for **RQ2.3**.

In this research question, we controlled the findings for size and number of changes, finding that none of them influence the outcome. We include a report of this additional analysis in our on-line appendix [14].

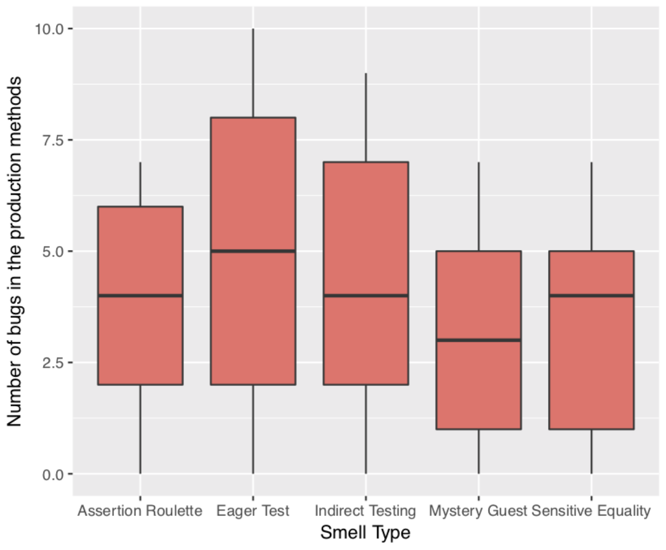

Fig. 9. Number of defects for different types of smells.

Finding 5. *The co-occurrence of more test smells in a test case is not strongly associated with higher defect-proneness of the exercised production code.*

RQ2.3 Are certain test smell types more associated with the defect-proneness of production code?

Figure 9 depicts the box plots reporting the association of different test smell types to the defect-proneness of production code. We observed that the 'Indirect Testing' and 'Eager Test' smells are related to the production code being more defect-prone with respect to the other test smell types. The differences observed between the 'Indirect testing' and 'Eager Test' and the other distributions are all statistically significant ($p-value < 2^{e-16}$) with medium effect size, while we found the other smells to be not statistically associated with more production code defect-proneness.

As also explained in the context of **RQ1.3**, the 'Indirect Testing' and 'Eager Test' smells lead to test cases that are (i) less cohesive and (ii) poorly focused on the target production code [74]. The former implies the testing of other objects indirectly, the latter checks several production methods of the class under test. The *lack of focus* of such smells may explain why the corresponding production code is associated with defect-proneness: It seems reasonable to consider that the *greedy* nature of these two smells makes them less able to find defects in the exercised production code.

From a practical point of view, our results provide evidence that developers should carefully monitor test and production code involved with *Indirect Testing* and *Eager Test*. In fact, these are the smells that not only are related to more change- and defect-prone test code, but also to more defect-prone production code.

Finding 6. *'Indirect Testing' and 'Eager Test' smells are associated with higher defect-proneness in the exercised production code. A likely motivation is the lack of focus of the tests on the target production code.*

VI. Conclusion

Automated testing is nowadays considered to be an essential process for improving the quality of software systems [12], [47]. Unfortunately, past literature showed that test code can often be of low quality and may contain design flaws, also known as test smells [7], [73], [74]. In this paper, we presented an investigation on the relation between six test smell types and test code change/defect proneness on a dataset of more than a million test cases. Furthermore, we delved into the relation between smelly tests and defect-proneness of the exercised production code.

The results we obtained provide evidence toward several findings, including the following two lessons:

Lesson 1. *Test smells and their relation with test code quality.* Corroborating what van Deursen *et al.* [74] conjectured in their study, we bring empirical evidence that test smells are negatively associated with test code quality. Specifically, we found that a smelly test has an 81% higher risk of being defective than a non-smelly test. Similarly, the risk of being change-prone is 47% higher in tests affected by smells. This result is complementary to the findings by Bavota *et al.* [7], who found that test smells can have a negative impact on program comprehension during maintenance activities. Moreover, we found that test methods with more, co-occurring smells tend to be more change-prone than methods having fewer smells and that 'Indirect Testing', 'Eager Test', and 'Assertion Roulette' are those associated with the most change-prone test code.

Lesson 2. *Test smells and their relation with software quality.* With our study, we provided empirical evidence that the presence of design flaws in test code is associated with the defect-proneness of the exercised production code; indeed the production code is 71% more likely to contain defects when tested by smelly tests. 'Indirect Testing' and 'Eager Tests' are related to a higher defect-proneness in production code.

This paper provides initial evidence on the relation between test smells and both change/defect proneness of test code and defect-proneness of exercised production code. As such, it represents a call to arms to researchers and tool vendors. We call upon researchers *and* tool vendors to develop practically automatic test smell detection tools. We call upon the research community to further investigate the interplay between test design quality and the effectiveness of test code in detecting defects.

VII. Acknowledgment

This project has received funding from the European Union's H2020 programme under the Marie Sklodowska-Curie grant agreement No 642954. A. Bacchelli and F. Palomba gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

References

[1] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 181–190. IEEE Computer Society, 2011.

[2] M. Aniche. Repodriller. https://github.com/mauricioaniche/repodriller, 2012.

[3] R. Arcoverde, A. Garcia, and E. Figueiredo. Understanding the longevity of code smells: preliminary results of an explanatory survey. In *Proceedings of the International Workshop on Refactoring Tools*, pages 33–36. ACM, 2011.

[4] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40(11):1100–1125, 2014.

[5] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[6] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[7] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. Are test smells really harmful? An empirical study. *Empirical Software Engineering*, 20(4):1052–1094, aug 2015.

[8] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[9] M. Beller, G. Georgios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman. Developer testing in the IDE: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*. To appear.

[10] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 179–190. ACM, 2015.

[11] S. Berner, R. Weber, and R. K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 571–579. ACM, 2005.

[12] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[13] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 53–62. ACM, 2011.

[14] BLINDED. Blinded. http://www.mediafire.com/?7pjmfjl2p1pmq, 2017.

[15] B. Boehm, D. H. Rombach, and M. V. Zelkowitz. *Foundations of Empirical Software Engineering*. Springer Berlin Heidelberg, 2005.

[16] G. Candea, S. Bucur, and C. Zamfir. Automated software testing as a service. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 155–160. ACM, 2010.

[17] D. N. Card and W. W. Agresti. Measuring software design complexity. *Journal of Systems and Software*, 8(3):185–197, 1988.

[18] I. Cunningham & Cunningham. Refactoring test code. http://wiki.c2.com/?RefactoringTestCode.

[19] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2017.

[20] M. D'Ambros, A. Bacchelli, and M. Lanza. On the impact of design flaws on software defects. In *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*, pages 23–31, 2010.

[21] F. A. Diaz-Quijano. A simple method for estimating relative risk using logistic regression. *BMC medical research methodology*, 12(1):14, 2012.

[22] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE, 2003.

[23] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. *Xtemp01*, pages 1–337, 1999.

[24] M. Gatrell and S. Counsell. The effect of refactoring on change and fault-proneness in commercial c# software. *Science of Computer Programming*, 102(0):44 – 56, 2015.

[25] M. Greiler, A. van Deursen, and M. A. Storey. Automated detection of test fixture strategies and smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 322–331, March 2013.

[26] M. Greiler, A. Zaidman, A. van Deursen, and M.-A. Storey. Strategies for avoiding text fixture smells during software evolution. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, pages 387–396. IEEE, 2013.

[27] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[28] M. R. Hess and J. D. Kromrey. Robust Confidence Intervals for Effect Sizes: A Comparative Study of Cohen's d and Cliff's Delta Under Non-normality and Heterogeneous Variances. *American Educational Research Association, San Diego*, nov 2004.

[29] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, June 2013.

[30] F. Khomh, M. Di Penta, Y. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on software changeability. *École Polytechnique de Montréal, Tech. Rep. EPM-RT-2009-02*, 2009.

[31] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, pages 75–84. IEEE Computer Society, 2009.

[32] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[33] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *Proceedings of the 9th International Conference on Quality Software*, pages 305–314, Hong Kong, China, 2009. IEEE CS Press.

[34] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[35] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE software*, 12(4):52–62, 1995.

[36] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[37] H. Li, J. Li, L. Wong, M. Feng, and Y.-P. Tan. Relative risk and odds ratio: A data mining perspective. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 368–377, New York, NY, USA, 2005. ACM.

[38] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, pages 1120–1128, 2007.

[39] A. Lozano, M. Wermelinger, and B. Nuseibeh. Assessing the impact of bad smells using historical information. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, IWPSE '07, pages 31–34, New York, NY, USA, 2007. ACM.

[40] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

[41] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*, pages 350–359. IEEE Computer Society, 2004.

[42] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison Wesley, 2007.

[43] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[44] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, page 13, 2001.

[45] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In T. Mens and S. Demeyer, editors, *Software Evolution*, pages 173–202. Springer, 2008.

[46] M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *Proceedings of the* 11^{th} *International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.

[47] G. Myers. *The Art of Software Testing, Second edition*, volume 15. 2004.

[48] S. M. Olbrich, D. Cruzes, and D. I. K. Sjøberg. Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10, 2010.

[49] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc. Numerical signatures of antipatterns: An approach based on b-splines. In R. Capilla, R. Ferenc, and J. C. Dueas, editors, *Proceedings of the* 14^{th} *Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, March 2010.

[50] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, pages 1–34, 2017.

[51] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *In Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110. IEEE, 2014.

[52] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Mining version histories for detecting code smells. *Software Engineering, IEEE Transactions on*, 41(5):462–489, May 2015.

[53] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pages 5–14. ACM, 2016.

[54] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman. A textual-based technique for smell detection. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, May 2016.

[55] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*, 2017.

[56] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *Proceedings of the International Conference on Software Maintenance (ICSME)*, pages 1–12. IEEE, 2017.

[57] F. Palomba, A. Zaidman, and A. D. Lucia. Automatic test smell detection using information retrieval techniques. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018.

[58] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia. An Exploratory Study on the Relationship between Changes and Refactoring. *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 176–185, 2017.

[59] R. Peters and A. Zaidman. Evaluating the lifespan of code smells using software repository mining. In *European Conference on Software Maintenance and ReEngineering*, pages 411–416. IEEE, 2012.

[60] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software*, 88:147–168, 2014.

[61] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu. Using history information to improve design flaws detection. In *8th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 223–232. IEEE Computer Society, 2004.

[62] D. Romano and M. Pinzger. Using source code metrics to predict change-prone java interfaces. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 303–312. IEEE, 2011.

[63] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol. An empirical study of code smells in javascript projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 294–305, Feb 2017.

[64] C. O. Schmidt and T. Kohlmann. When to use the odds ratio or the relative risk? *International journal of public health*, 53(3):165–167, 2008.

[65] D. J. Sheskin. Handbook of Parametric and Nonparametric Statistical Procedures. *Technometrics*, 46(3):369–370, aug 2004.

[66] D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba. Quantifying the effect of code smells on maintenance effort. *Software Engineering, IEEE Transactions on*, 39(8):1144–1156, Aug 2013.

[67] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005*. ACM, 2005.

[68] H. M. Sneed. Reverse engineering of test cases for selective regression testing. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 69–74. IEEE, 2004.

[69] P. Thongtanunam, S. Mcintosh, A. E. Hassan, and H. Iida. Review participation in modern code review - An empirical study of the android, Qt, and OpenStack projects. *Empirical Software Engineering (EMSE)*, 22(2):768–817, 2017.

[70] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[71] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 4–15. ACM, 2016.

[72] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *Transactions on Software Engineering (TSE)*, 43(11):1063–1088, 2017.

[73] A. Vahabzadeh and A. Mesbah. An Empirical Study of Bugs in Test Code. pages 101–110, 2015.

[74] A. van Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, pages 92–95, 2001.

[75] B. Van Rompaey and S. Demeyer. Establishing traceability links between unit test cases and units under test. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 209–218. IEEE, 2009.

[76] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, Dec 2007.

[77] F. Wilcoxon. Individual comparisons of grouped data by ranking methods. *Journal of economic entomology*, 39(6):269, 1946.

[78] A. Yamashita and L. Moonen. Do code smells reflect important maintainability aspects? In *Proc. Int'l Conf. on Software Maintenance (ICSM)*, pages 306–315. IEEE, 2012.

[79] A. Yamashita and L. Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 682–691. IEEE, 2013.

[80] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 121–130. IEEE Computer Society, 2015.

[81] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen. Mining Software Repositories to Study Co-Evolution of Production & Test Code. In *2008 International Conference on Software Testing, Verification, and Validation*, volume 3, pages 220–229. IEEE, apr 2008.

[82] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[83] J. Zhang and F. Y. Kai. What's the relative risk?: A method of correcting the odds ratio in cohort studies of common outcomes. *Jama*, 280(19):1690–1691, 1998.

[84] Y. Zhou, H. Leung, and B. Xu. Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. *IEEE Transactions on Software Engineering*, 35(5):607–623, 2009.